Evaluating the Effect of Coherence Protocols on the Performance of Parallel Programming Constructs *

Ricardo Bianchini, Enrique V. Carrera E., and Leonidas Kontothanassis†

COPPE Systems Engineering
Federal University of Rio de Janeiro
{ricardo,vinicio}@cos.ufrj.br

†Cambridge Research Laboratory
Digital Equipment Corporation
kthanasi@crl.dec.com

Abstract

The different implementations of parallel programming constructs interact heavily with a multiprocessor's coherence protocol and thus may have a significant impact on performance. The form and extent of this interaction have not been established so far however, particularly in the case of update-based coherence protocols. In this paper we study the running time and communication behavior of (1) ticket and MCS spin locks, (2) centralized, dissemination, and tree-based barriers, (3) parallel and sequential reductions, (4) linear broadcasting and producer and consumer-driven logarithmic broadcasting, and (5) centralized and distributed task queues, under pure and competitive update coherence protocols on a scalable multiprocessor; results for a write invalidate protocol are presented mostly for comparison purposes. Our experiments indicate that parallel programming techniques that are well-established for write invalidate protocols, such as MCS locks and task queues, are often inappropriate for update-based protocols. In contrast, techniques such as dissemination and tree barriers achieve superior performance under update-based protocols. Our results also show that update-based protocols sometimes lead to different design decisions than write invalidate protocols. Our main conclusion is that indeed the interaction of the parallel programming constructs with the multiprocessor's coherence protocol has a significant impact on performance. The implementation of these constructs must be carefully matched to the coherence protocol if ideal performance is to be achieved.

Keywords: Parallel Constructs, Coherence Protocols, Scalable Multiprocessors, Performance Evaluation.

*This research was supported by Brazilian National Research Council (CNPq).

1 Introduction

Some of the most common parallel programming idioms are locks, barriers, task queues, and broadcast and reduction operations. The different implementations of these constructs interact heavily with a multiprocessor's cache coherence protocol and thus may have a significant impact on performance. The form and extent of this interaction have not been established so far however, particularly in the case of update-based coherence protocols.

Past studies of update-based protocols for cache-coherent multiprocessors (e.g. [3, 6]) have ultimately focused on overall application performance in order to evaluate these protocols. Studies of multiprocessor communication behavior (e.g. [11, 7, 5]) also tend to concentrate on the overall application behavior, without isolating the behavior of the different parallel programming constructs and techniques used by the applications.

While early studies focused on overall trends, in this work we isolate the behavior of individual parallel constructs under different cache-coherence protocols. Our main goals are to evaluate the performance and communication traffic of implementation techniques that are well-established for write invalidate (WI) protocols under update-based coherence and to compare different implementation/protocol combinations on a scalable multiprocessor.

In particular we seek to understand the performance of various implementations of process synchronization, reduction and broadcasting operations, and load balancing strategies under pure update (PU) and competitive update (CU) coherence protocols; we also present WI results mostly for comparison purposes. We use execution-driven simulation of a 32-node scalable multiprocessor to study the performance of (1) ticket and MCS spin locks, (2) centralized, dissemination, and tree-based barriers, (3) parallel and sequential reduction operations, (4) linear broadcasting and producer and consumer-driven logarithmic broadcasting, and (5) centralized and distributed task queues. The execution time behavior of each combination of implementation and protocol is explained by the amount and usefulness of the communication generated by the combination.

We only consider software constructs in conjunction with invalidate and update-based coherence protocols. It is possible to create more efficient versions of some of our software constructs using additional hardware features such as prefetching instructions (e.g. [19]), remote writes (e.g. [1]), or messages in multiprocessors with protocol processors, but the study of such implementations is beyond the scope of this paper. We have restricted our evaluation to the three dominant coherence protocols and a host of parallel programming constructs and implementation techniques that leverage those protocols.

Our most interesting results show that for scalable multiprocessors:

- the ticket lock under the update-based protocols outperforms all other protocol/implementation combinations up to 4-processor machine configurations, while the MCS lock under CU performs best for larger numbers of processors;
- the standard MCS lock is inappropriate for a PU protocol, but a slight modification of this lock can improve its performance;
- dissemination and tree barriers perform significantly better under update-based protocols than under the WI protocol;
- the dissemination barrier under the update-based protocols is ideal for all numbers of processors;
- when processes are tightly-synchronized, sequential reductions outperform parallel ones, independently of the protocol used;
- overall, update-based sequential reductions exhibit best performance when processes are tightly synchronized;
- logarithmic broadcasting performs significantly better under update-based protocols than under the WI protocol;
- consumer-driven broadcasting under PU is ideal for all numbers of processors;
- consumer-driven broadcasting outperforms its producer-driven counterpart for WI and PU, but not for CU;
- the PU protocol is inappropriate for dynamic load balancing, even when it is implemented via distributed task queues;
- WI is the best protocol for both centralized and distributed task queues.

In summary, our results show that parallel programming techniques that are well-established for WI protocols, such as MCS locks, parallel reductions, and task queues, can become performance bottlenecks under update-based protocols. In contrast, techniques such as dissemination and tree barriers and logarithmic broadcasting achieve superior performance under update-based protocols.

Our results also show that update-based protocols sometimes lead to different design decisions than WI protocols, as demonstrated by our broadcasting experiments. Given the characteristics of the traffic generated by the different coherence protocols we study, update-based approaches are ideal for scalable barrier synchronization, sequential reductions, and logarithmic broadcasting, while WI is ideal for scalable lock synchronization and task queues.

Our main conclusion is that the interaction of the parallel programming constructs with the multiprocessor's coherence protocol has a significant impact on performance. The implementation of these constructs must be carefully matched to the coherence protocol if ideal performance is to be achieved. For multiprocessors with hard-wired coherence protocols (e.g. [16, 13, 2]), these implementations must match the native protocol, while for multiprocessors that can support multiple coherence protocols (e.g. [14, 20]), the best combination of implementation and protocol must be chosen.

The remainder of this paper is organized as follows. Section 2 presents the constructs and techniques we study and their implementations. Section 3 describes our methodology and performance metrics. Experimental results are presented in section 4. Section 5 summarizes the related work. Finally, section 6 summarizes our findings and concludes the paper.

2 Parallel Programming Constructs and Techniques

Parallel programming for multiprocessors involves dealing with such issues as lock and barrier synchronization and reduction operations. These aspects of parallel applications can be implemented in various ways, the most important of which we describe in this section.

2.1 Spin Locks

Spin locks are extremely common parallel programming constructs. We will consider two types of spin locks: the ticket lock with proportional backoff and the MCS list-based queuing lock [17]. We chose to study these types of locks as previous studies of several lock implementations under WI protocols have shown that the ticket lock is ideal for low-contention scenarios, while the MCS lock performs best for highly-contended locks.

As seen in figure 1, the ticket lock employs a global counter that provides "tickets" determining when the processor will be allowed to enter the critical section. Another global counter determines which ticket is currently being serviced. A processor is allowed to acquire the lock when its ticket is the same as indicated by the service counter. Whenever this is not the case, the processor pauses (spins locally) for a time proportional to the difference between its ticket and the counter, in order to reduce contention for the counter.

The basic idea behind the MCS lock (figure 2) is that processors holding or waiting for access to the lock are chained together in a list. Each processor holds the address of the processor behind it in the list. Each waiting processor spins on its own Boolean flag. The processor releasing a lock is responsible for removing itself from the list and changing the flag of the processor behind it.

Although MCS locks can be efficient under WI, they may generate a large amount of traffic under update-based protocols as all processors competing for a lock may end up caching all other processors' I variables and receiving an update for each modification of these variables. In order to avoid this problem, we propose a modification to the MCS lock in which a processor flushes the I's of its predecessor and successor in the list. The flush operation can be implemented using the user-level block flush instruction common to modern microprocessors such as the PowerPC 604. The blocks to be flushed in the modified MCS lock are indicated with comments in figure 2.

2.2 Barriers

Just like spin locks, barriers are common parallel programming constructs. We study three different types of barriers: the sense-reversing centralized barrier, the dissemination barrier, and the tree-based barrier proposed in [17]. We chose to consider these types of barriers as previous studies of synchronization under WI protocols have suggested that centralized barriers are very good for small-scale multiprocessors, while dissemination and tree-based barriers are ideal for large-scale multiprocessors.

In the sense-reversing centralized barrier (figure 3) each processor decrements a variable counting the number of processors that have already reached the barrier. The sense variable prevents a processor from completing two consecutive barrier episodes without all processors having completed the first one.

Several algorithms have been proposed to avoid the centralized nature of this barrier. An efficient one, the dissemination barrier (figure 4), replaces a single global synchronization event with $\lceil \log_2 P \rceil$ rounds of synchronizations with a specific pattern; in round k, processor i signals processor $(i + 2^k) \bmod P$, where P is the number of processors. Interference between consecutive barrier episodes is avoided by using alternating sets of variables.

Another efficient distributed barrier algorithm is the tree-based barrier proposed in [17]. This algorithm uses an arrival tree where each group of 4 processors signals barrier arrival events to their common parent, and a wake-up flag to notify the completion of a barrier episode. Pseudo-code for this algorithm is presented in figure 5.

2.3 Reductions

Reduction operations are used in parallel programs in order to produce a "global" result out of "local" arguments. Reductions usually apply a specific operator, such as *min* or *sum*, to per processor arguments to produce a machine-wide result. Sometimes these arguments are themselves produced by several local applications of the operator.

Reductions can be performed in parallel or sequentially. In a sequential reduction, one processor is responsible for computing the global value sequentially. An example sequential reduction operation to compute the overall sum of each processor's local sum is presented in figure 6. This type of reduction is necessary for multiprocessors that lack a fetch_and_add instruction. In a parallel reduction, all processors modify a global variable themselves inside a critical section, as seen in figure 7. Note that the structure of the implementation presented in the figure can frequently be found in parallel programs, but is only appropriate for relatively small numbers of processors. A tree-like implementation of a parallel reduction should perform better than our implementation for large numbers of processors.

One might wonder why sequential reductions are even reasonable. Two important aspects of parallel and sequential reductions may shed some light into this issue: (1) when processors are tightly synchronized in the parallel reduction, the critical path of the algorithm includes the sum of the critical sections of all processors that queue up for the lock; and (2) due to the manipulation of the lock variable, the sum of P critical sections of the parallel reduction is much longer than the critical path of the sequential reduction (according to our careful analysis of the code generated by gcc with -O2 optimization level).

2.4 Broadcasting

Broadcast operations are required in parallel programs whenever data produced by a processor must be accessed by all other processors. Broadcasting can be performed in linear or logarithmic (tree-like) fashion. In linear broadcasting, all consumers access the produced data directly, which leads to the simple code in figure 8 but can generate excessive contention for large machine configurations. Several buffers may be used to allow for varying production and consumption rates and thus permitting different broadcasts to overlap.

Logarithmic broadcasting can be used to prevent significant contention on large numbers of processors. A logarithmic broadcast operation can be implemented in a producer or consumer-driven fashion. In its producer-driven version, the producer copies the data to its children in the tree, which in turn copy the data to their children and so on. In its consumer-driven version, when data are produced by the producer, the producer's children copy the data initially, which prompts their children to copy data from parents and so on. The consumer and producer-driven logarithmic broadcast operations we study are shown in figures 9 and 10, respectively. Note that all the broadcasting codes require fences to ensure correct execution under the memory consistency model we assume, release consistency.

2.5 Task Queues

Task queues are frequently used in parallel programs for dynamic load balancing, as in the Cholesky application from the Splash2 suite [24]. Dynamic load balancing is necessary when the amount of load assigned to each processor is a function of the application's input or when new and unpredictable work is created during run time.

The basic idea behind the task queue is to collect descriptors of work that is yet to be done. Processors dequeue descriptors whenever they become idle and enqueue descriptors when they produce new work. Task queues can be implemented in either centralized or distributed form. Figure 11 presents the pseudo-code of the centralized task queue example kernel we study. The kernel uses a global counter to represent the task queue. Processors dequeue (fetch_and_increment the counter) when they need work; there is no need for enqueueing work descriptors, since each value of the counter describes the piece of work to be performed. In order to avoid an excessive number of accesses to the global counter, several tasks are associated with each work descriptor. All tasks are executed TOT_ROUNDS times.

Figure 12 presents the pseudo-code of the distributed task queue example kernel we study. The kernel uses per processor counters containing all the work to be performed in each round. Each processor initially dequeues work from its local counter and, when all its local tasks have been exhausted, searches for work in other counters. A processor only takes work away from a remote processor if it detects that the remote processor is far behind in its computation. In order to avoid an excessive number of accesses to remote counters, an idle processor will only search for work on 1/4 of the remote processors. Again, all tasks are executed TOT_ROUNDS times.

3 Methodology

We are interested in assessing and categorizing the communication behavior of our construct implementations under different multiprocessor coherence protocols and, therefore, we use simulation for our studies.

3.1 Multiprocessor Simulation

We use a detailed execution-driven simulator (based on the MINT front-end [23]) of a 32-node, DASH-like [16], mesh-connected multiprocessor. Each node of the simulated machine contains a single processor, a 4-entry write buffer, a 64-KB direct-mapped data cache with 64-byte cache blocks, local memory, a full-map directory, and a network interface. Shared data are interleaved across the memories at the block level. All instructions and read hits are assumed to take 1 cycle. Read misses stall the processor until the read request is satisfied. Writes go into the write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. A memory module can provide the first word 20 processor cycles after the request is issued. Other words are delivered at a rate of 1 word per processor cycle. Memory contention is fully modeled. The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor clock speed. Switch nodes introduce a 2-cycle delay to the header of each message. The network datapath is 16-bit wide. Network contention is only modeled at the source and destination of messages.

Our WI protocol keeps caches coherent using the DASH protocol with release consistency [15]. In our update-based implementations, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. The writing processor only stalls waiting for acks at a lock release point.

Our PU implementation includes two optimizations. First, when the home node receives an update for a block that is only cached by the updating processor, the ack of the update instructs the processor to retain future updates since the data is effectively private. Second, when a parallel process is created by *fork*, we flush the cache of the parent's processor, which eliminates useless updates of data written by the child but not subsequently needed by the parent.

In our CU implementation, each node makes a local decision to invalidate or update a cache block when it sees an update transaction. We associate a counter with each cache block and invalidate the block when the counter reaches a threshold. At that point, the node sends a message to the block's home node asking it not to send any more updates to the node. References to a cache block reset the counter to zero. We use counters with a threshold of 4 updates.

Our simulator implements three atomic instructions: fetch_and_add, fetch_and_store, and compare_and_swap. The coherence protocol used for the atomically-accessed data is always the same as the protocol used for all the rest of the shared data. The computational power of the atomic instructions is placed in the cache controllers when the coherence protocol is WI and in the memory when using an update-based protocol. So, for instance, the fetch_and_add instruction under the WI protocol obtains an exclusive copy of the referenced block and performs the addition locally. Fetch_and_add under an update-based protocol sends an addition message to the home memory, which actually performs the addition and sends update messages with the new value to all processors sharing the block. The atomic instructions we implement force all previous locally-issued read and write operations to have globally performed before taking action.

3.2 Performance Metrics

The focus of this paper is on running times and our categorization of the communication traffic in invalidate and update-based protocols. We consider the communication generated by cache misses (and block upgrade operations) under both types of protocol and the update messages under update-based protocols. The miss rate is computed solely with respect to shared references.

In order to categorize cache misses we use the algorithm described in [8] as extended in [4]. The algorithm classifies cache misses as cold start, true sharing, false sharing, eviction, and drop misses. We assume cold start and true sharing misses to be *useful* and the other types of misses to be *useless*. More specifically, the different classes of cache misses in our algorithm are:

- **Cold start misses.** A cold start miss happens on the first reference to a cache block by a processor.
- **True sharing misses.** A true sharing miss happens when a processor references a word belonging in a block it had previously cached, but that has been invalidated due to a write to the same word by some other processor.
- **False sharing misses.** A false sharing miss occurs in roughly the same circumstances as a

true sharing miss, except that the word written by the other processor is not referenced by the missing processor.

- **Eviction misses.** An eviction (replacement) miss happens when a processor replaces one of its cache blocks with another one mapping to the same cache line and later needs to reload the block replaced.

- **Drop misses.** A drop miss happens when a processor references a word belonging in a block it had previously cached, but that has been self-invalidated under the competitive update protocol.

Our miss categorization algorithm includes a sixth category, **exclusive** request transactions. An exclusive request operation (caused by a write to a read-shared block already cached by the writing processor under the WI protocol) is not strictly a cache miss, but does cause communication traffic.

We classify update messages according to the algorithm described in [4]. The algorithm classifies update messages at the end of an update's lifetime, which happens when it is overwritten by another update to the same word, when the cache block containing the updated word is replaced, or when the program ends. We also classify updates as useful and useless. Intuitively, useful updates are those updates required for correct execution of the program, while useless updates could be eliminated entirely without affecting the correctness of the execution. More specifically, the different classes of updates in our algorithm are:

- **True sharing updates.** The receiving processor references the word modified by the update message before another update message to the same word is received. This type of update transaction is termed useful, since it is necessary for the correctness of the program.

- **False sharing updates.** The receiving processor does not reference the word modified by the update message before it is overwritten by a subsequent update, but references some other word in the same cache block.

- **Proliferation updates.** The receiving processor does not reference the word modified by the update message before it is overwritten, and it does not reference any other word in that cache block either.

- **Replacement updates.** The receiving processor does not reference the updated word until the block is replaced in its cache.

- **Termination updates.** A termination update is a proliferation update happening at the end of the program.

- **Drop updates.** A drop update is the update that causes a block to be invalidated in the cache.

This categorization is fairly straightforward, except for our false update class. Successive (useless) updates to the same word in a block are classified as proliferation instead of false sharing updates, if the receiving processor is not concurrently accessing other words in the block. Thus, our algorithm classifies useless updates as proliferation updates, unless *active* false sharing is detected or the application terminates execution.

3.3 Experiments

Section 4 presents running time and communication performance results of several implementations of each programming construct we study running on our simulator. All our experiments focus on synthetic rather than real programs. Synthetic programs allow us to isolate the behavior of each implementation under the different coherence protocols when there is an actual difference between the protocols, i.e. when the shared data used by our implementations remain in the processors' caches. Real programs would provide for more realistic results but could interfere with our comparison by causing the eviction of important shared variables from the caches. An unfortunate side effect of our experimental methodology is however that it may overstate both the effectiveness of update-based protocols in reducing the number of cache misses and the performance degradation resulting from useless update messages.

4 Experimental Results

In this section we evaluate the performance of the different implementations of locks, barriers, reductions, broadcasts, and task queues under the three coherence protocols we consider. In all implementations, shared data are mapped to the processors that use them most frequently.

4.1 Spin Locks

In order to assess the performance of each combination of spin lock implementation and coherence protocol under varying levels of lock contention, we developed a synthetic program where each processor acquires a lock, spends 50 processor cycles holding it, and then releases the lock. All of this in a tight loop executed $32000/P$ times, where P is the number of processors.

Figure 13 presents the average latency of an acquire-release pair (in processor cycles) for each machine configuration. This average latency is computed by taking the execution time of the synthetic program, dividing it by 32000, and later subtracting 50 from the result. Figure 14 presents the number and distribution of cache misses involved in each of the lock/protocol combinations on 32 processors, while figure 15 presents the number and distribution of update messages in the lock implementations using the update-based protocols again on 32 processors [1]. The labels in these figures represent the specific algorithm/protocol combinations: "tk", "MCS", and "uc" stand for ticket, MCS, and update-conscious MCS locks respectively, while "i", "u", and "c" stand for WI, PU, and CU respectively. Note that we placed the absolute numbers of cache misses on top of the bars that are not tall enough to be noticeable.

For ticket locks, figure 13 shows that CU performs slightly better than PU for 32 processors, while both protocols perform significantly better than WI for all machine configurations. As seen in figures 14 and 15, the reason for this result is that the update-based protocols exchange the expensive cache misses necessary to constantly re-load the `ticket` and `now` counters in WI, for corresponding update messages that only lead to performance degradation if they end up causing resource contention.

For the MCS lock, the CU protocol outperforms all other combinations for 16 and 32 processors; trends indicate that for larger numbers of processors the WI protocol should become best. The MCS lock exhibits terrible performance under PU; the implementation using this protocol is worse than the ones with WI and CU by a factor of 2 for 32 processors. The main problem with the MCS lock under PU protocols is that it increases the amount of sharing (by sharing the global pointer to the end of the list and pointers to list predecessors and successors) with respect to a ticket lock, without reducing the frequency of write operations on the shared data. This increased sharing causes intense messaging activity (proliferation updates mostly) that degrades performance, as seen in figure 15.

Our modification to the MCS lock significantly alleviates the sharing problem of the standard MCS lock under PU protocols, as seen by the 39% reduction in update messages the modification produces. However, much of the effect of this reduction is counter-balanced by an increase in cache

[1]The bars in the figure do not include the replacement updates category since this type of updates was only observed in our broadcasting and task queue experiments.

miss activity from 1089 to 31588 misses. The outcome of this tradeoff is 18% and 11% performance improvements for 16 and 32 processors, respectively. Note that the extent to which the reductions in traffic provided by our update-conscious MCS lock lead to performance improvements depends on the architectural characteristics of the multiprocessor: performance improvements are inversely proportional to communication bandwidth and latency.

Overall, we find that within the range of machine sizes we consider ticket locks with update-based protocols achieve best performance up to 4 processors, while MCS locks under CU are ideal for larger numbers of processors. Our update-conscious implementation of MCS locks improve the performance of this lock algorithm, but not enough to justify its use when a choice is available. Finally, we also find that, independently of the lock implementation, the vast majority of updates under an update-based protocol is useless.

These experiments were purposedly made similar to the ones performed by Mellor-Crummey and Scott in [17] for comparison purposes. We also performed experiments with a slightly modified synthetic program where, instead of releasing the lock and immediately trying to grab it again, processors waste a pseudo-random (but bounded) amount of time after the release. This modified synthetic program provides for reduced lock contention. The results of these modified experiments are qualitatively the same as the ones presented in this section. In a more controlled experiment, we made the ratio of work outside and inside the critical section be equal to the number of processors (+- 10%). Again, the results of this modified experiment are qualitatively similar to the ones discussed in this section.

4.2 Barriers

In order to assess the performance of each combination of barrier implementation and coherence protocol, we developed a synthetic program where processors go through a barrier in a tight loop executed 5000 times.

Figures 16 presents the execution time (in processor cycles) of the synthetic program running on different numbers of processors divided by 5000. This time is thus the average latency of a barrier episode for each machine configuration. Figure 17 presents the cache miss behavior of each of the barrier/protocol combinations on 32 processors, while figure 18 presents the update behavior of the barrier implementations using the update-based protocols again on 32 processors. The labels in these figures represent the specific algorithm/protocol combinations: “cb”, “db”, and “tb” stand for centralized, dissemination, and tree-based barriers respectively.

Figure 16 shows that for centralized barriers the WI protocol outperforms its update-based counterparts, but only for large machine configurations, as one would expect. Figures 17 and 18 show that even though the number of misses of the centralized barriers under the update-based protocols is negligible, the amount of update traffic these protocols generate is substantial and mostly useless. The vast majority of this useless traffic corresponds to changes in the centralized counter of barrier arrivals.

Before moving on to the scalable barriers, it is interesting to note that there is not a significant difference between the amount of update traffic generated by PU and CU for the centralized barrier. The number of update messages under PU is exactly what one would expect: $\mathcal{O}(P^2)$, where P is the number of processors. However, the update traffic under CU should be much less intense than shown in the figure; there should only be $\mathcal{O}(P)$ updates. The only reason why our results do not match this intuition is that processors are somewhat tightly synchronized in our synthetic program and, as a consequence, the messages requesting elimination from the sharing set of the block that contains the counter end up queued behind the fetch_and_decrement messages at the block's home node.

For the dissemination barrier CU and PU perform equally well, significantly outperforming WI for all numbers of processors. The reason for this result is that WI causes a relatively large number of cache misses on accesses to the `myflags` array, while the update behavior of the dissemination barrier under CU and PU is very good (as demonstrated by their lack of useless update messages). In essence, these protocols behave well as a result of the fixed, low-degree, and write-once sharing pattern exhibited by the dissemination barrier.

For the tree-based barrier PU and CU again (and for the same reasons) perform equally well and much better than WI for all numbers of processors.

These results indicate that the dissemination barrier under either PU or CU is the combination of choice for all numbers of processors. In addition, our barrier results demonstrate that update-based protocols perform extremely well for scalable barriers, as shown by the absence of useless update messages in these barriers.

4.3 Reductions

In order to assess the performance of each combination of reduction implementation and coherence protocol, we developed a synthetic program where each processor executes 5000 reductions in a tight loop. To avoid disturbing the results of our reduction experiments, we simulated (fake or idealized) lock and barrier operations that generate the appropriate network traffic, provide the correct synchronization behavior, but that do not execute real synchronization algorithms, i.e. processor instructions.

Figure 19 presents the execution time (in processor cycles) of the synthetic program running on different numbers of processors divided by 5000. This time is thus the average latency of a whole reduction operation for each machine configuration. Figure 20 presents the cache miss behavior of each of the reduction/protocol combinations on 32 processors, while figure 21 presents the update behavior of the reduction implementations using the update-based protocols again on 32 processors. The labels in these figures represent the specific algorithm/protocol combinations: "sr" and "pr" stand for sequential and parallel reductions respectively.

Figure 19 shows that the sequential reduction outperforms its parallel counterpart for all protocols and numbers of processors. One effect that contributes to this result is that for our tightly-synchronized synthetic program, the critical path of the parallel reduction is longer (as explained in section 2.3). This is the overriding effect for WI, where sequential and parallel reductions exhibit the same number and type of cache misses as shown in figure 20. For the update-based protocols other factors contribute to the superiority of sequential reductions: for CU, parallel reductions entail an excessive number of cache misses (drop misses on accesses to `sum`), while for both PU and CU parallel reductions lead to a large amount of useless update traffic. Figure 21 shows that sequential reductions exhibit a large percentage of useful updates, indicating that update-based protocols are appropriate for this type of operation, just as they are for scalable barriers.

A comparison between update-based and invalidate-based reductions is also interesting. Overall, update-based sequential reductions always exhibit better performance than sequential reductions under WI. The reason for this result is that the critical path of a sequential reduction under WI is significantly longer than the critical path of a sequential reduction under the update-based protocols due to the cache misses involved in the former algorithm/protocol combination. Note that up to 16 processors even parallel reductions under PU perform better than sequential reductions under WI.

Although interesting, these experiments only model the case where processes are tightly synchronized and most processors end up contending for lock access. We also performed experiments with a slightly modified synthetic program to generate some load imbalance and consequently reduce lock contention. The results of these experiments show that parallel reductions become more efficient than their sequential counterparts, but still parallel reductions with PU and CU perform better than parallel reductions with WI.

4.4 Broadcasting

In order to assess the performance of each combination of broadcasting implementation and coherence protocol, we developed a synthetic program where 64 bytes of data are broadcast 5000 times in a tight loop. 8 buffers are used to permit the overlap of different broadcast operations. The degree of the broadcast tree is 2 in our logarithmic broadcasting experiments. In general, wider trees would generate higher memory and network contention in the consumer-driven broadcast and higher serialization in the producer-driven broadcast. Narrower trees alleviate these problems, but increase the time it takes a produced item to reach the last of its consumers.

Figure 22 presents the execution time (in processor cycles) of the synthetic program running on different numbers of processors divided by 5000. This time is thus the average latency of a whole broadcast operation for each machine configuration. Figure 23 presents the cache miss behavior of each of the broadcasting/protocol combinations on 32 processors, while figure 24 presents the update behavior of the broadcasting implementations using the update-based protocols again on 32 processors. The labels in these figures represent the specific algorithm/protocol combinations: “ln”, “cs” and “pd” stand for linear, consumer and producer-driven logarithmic broadcasting respectively.

The results in figure 22 show that except for very small machine configurations (up to 4 processors), the linear broadcast technique suffers severely from contention effects. These effects are particularly harmful to performance for the update-based protocols, where a tremendous number of update messages generated by the producer (when it writes to data buffers and to indices) and consumers (when they fetch_and_decrement the counters) congest the network.

The figure also shows that PU and CU perform exactly the same for the producer-driven broadcast. The reason is that the number of consecutive writes without intervening local references (1) to each piece of shared data is always smaller than our competitive threshold (4). The update-based protocols significantly outperform WI for the producer-driven broadcast on all numbers of processors (except 1 of course). The main reason for this result is that, during the broadcast phase, most accesses to the `broad` array cause cache misses. Figure 23 demonstrates that the miss rate of the producer-driven broadcast under WI is more than a factor of 10 higher than under the update-based protocols. Furthermore, as seen in figure 24, the update traffic generated by the producer-driven broadcast is 100% useful.

The update-based protocols also outperform WI for the consumer-driven broadcast, again mostly as a result of the poor WI cache behavior in accesses to the `broad` array, during the broadcast phase. In contrast with the producer-driven broadcast however, the performance of the consumer-driven broadcast is worse under CU than PU, since the number of consecutive writes without local references to `counter` variables often exceeds our competitive threshold.

Comparing the two implementations of logarithmic broadcasting, we find that the consumer-driven broadcast under PU represents the best implementation/protocol combination overall, for all numbers of processors, even though its update traffic behavior is significantly worse than for the producer-driven/PU combination. The reason for this result is the longer critical path of the producer-driven broadcast, which includes the sequential copies of data from each parent to its children.

It is also interesting to observe that the consumer-driven broadcast is more efficient than its producer-driven counterpart under WI and PU, but not under the CU protocol. For CU the higher miss rate and update traffic of the consumer-driven broadcast lengthen the critical path of the algorithm beyond that of the producer-driven broadcast.

4.5 Task Queues

In order to assess the performance of each combination of our task queue-based computation and coherence protocol, we developed a synthetic program where a 256×256 matrix is completely re-written in parallel 50 times. Each work descriptor (counter value) is associated with the modification of `256 / P / 2` rows of the matrix in the centralized implementation and 1 row of the matrix in the distributed implementation, where `P` is the number of processors. Note that in contrast with the centralized task queue implementation, the distributed queue kernel does not perform any chunking of tasks, i.e. it associates a single task with each work descriptor. The reason for this difference is that chunking is not as important when there is little contention for task queues. Thus, avoiding chunking in the distributed task queue kernel allows us to control the load balancing finely, without incurring significant overhead.

In order to generate some load imbalance in the synthetic program, the work associated with a descriptor is only performed with 50% probability. The value of the `BEHIND` parameter of the distributed queue implementation was picked to be 3 based on the ratio of the costs of performing a task with local data and performing the task with remote data: roughly 5300 cycles/roughly 9000 cycles, i.e. it is about 70% more expensive to perform the task with remote data.

Figure 25 presents the execution time (in processor cycles) of the synthetic program running on different numbers of processors divided by 50. This time is thus the average latency of a whole parallel modification of the matrix for each machine configuration. Figure 26 presents the cache miss behavior of each of the task queue/protocol combinations on 32 processors, while figure 27 presents the update behavior of the task queue implementations using the update-based protocols again on 32 processors. The labels in these figures represent the specific implementation/protocol combinations: "cent" stands for centralized and "dist" stands for distributed task queues. To avoid disturbing the results of our task queue experiments, we simulated idealized synchronization operations that generate the appropriate network traffic, provide the correct synchronization behavior, but that do not execute real synchronization algorithms, i.e. processor instructions.

As one would expect, figure 25 shows that the distributed task queue implementation outperforms its centralized counterpart for the three coherence protocols we study. For all protocols the performance difference between the two implementations decreases with machine size. For instance, we find that under WI centralized task queues are only 30% worse than distributed task queues for 32 processors. The decreasing performance difference is a consequence of the fact that, in the distributed task queue experiments, the number of tasks in the local pools becomes an ever smaller fraction of the total number of tasks as we increase the number of processors.

The figure also shows that PU achieves terrible performance for both centralized and distributed task queues. For centralized task queues, PU does not perform well due to the fact that there is no affinity between a row of the matrix and the processor where it was last modified, i.e. data sharing becomes more widespread with each new round of computation. For distributed task queues, PU exhibits poor performance for two reasons: a) the data associated with descriptors that are effectively migrated leave their footprint in the caches they visit, i.e. subsequent modifications to these data are sent to all these caches; and b) the data migrated is only useful once in most cases, since the amount and distribution of the load changes from one round of computation to the next. The CU protocol performs significantly better than PU for the two task queue implementations as a result of its better sharing behavior, as demonstrated in figure 27. In fact, CU and WI perform equally well for distributed task queues, outperforming all other implementation/protocol combinations. Note that WI only entails a factor of 4 more "real" cache misses [2] than PU and slightly more real misses than CU for distributed task queues, as shown in figure 26. This miss rate comparison shows that WI is closer to the update-based protocols for distributed task queues than for any other programming construct we study in this paper.

[2]Recall that exclusive requests do not really stall the processor, unless the write buffer is full.

5 Related Work

As far as we are aware, this study is the first to isolate the performance of important parallel programming constructs and techniques under PU and CU protocols. This analysis led to a number of interesting observations that challenge previously-established results. In addition, our study is the first to relate these constructs and techniques to their communication behavior under invalidate, update, or competitive protocols. Some related pieces of work are listed next.

The impact of coherence protocols on application performance is an active area of research. Early work by Eggers [10] studied the relative performance of invalidate and update protocols on small bus-based cache-coherent multiprocessors. More recent work [6, 22] has looked at the impact of update-based protocols on overall program performance on larger machines.

Other researchers have taken an applications-centric view and have focused on the communication patterns induced by applications, mostly in the context of WI protocols. Gupta and Weber [11] looked at the cache invalidation patterns in shared-memory multiprocessors, and determined that for most applications the degree of sharing is small. Holt et al. [12] also looked at the communication behavior of applications in the context of large-scale shared-memory multiprocessors and identified architectural and algorithmic bottlenecks. Dubois et al. [8], Torrellas at al. [21], and Eggers et al. [9] have looked at the usefulness of communication traffic generated by real applications in the context of WI protocols. Bianchini et al. [5] and Dubois et al. [7] have looked at the usefulness of communication traffic under both invalidate and update-based protocols.

Parallel programming constructs and in particular synchronization algorithms have also received a lot of attention, however almost always in the context of either non-coherent multiprocessors or machines based on WI protocols. Mellor-Crummey and Scott [17], for instance, have presented the set of synchronization algorithms that we have used in our evaluation of synchronization primitives. They evaluated these algorithms on top of a non-coherent multiprocessor and a bus-based multiprocessor with WI coherence. Among other interesting results, they showed that the communication architecture of the multiprocessor may significantly affect the comparison of different construct implementations. For example, they found that dissemination barriers are outperformed by tree-based barriers on bus-based machines, which is in contrast with their (and our) scalable multiprocessor results. Abdel-Shafi et al. [1] have studied MCS locks and tree-based barriers under WI but in the presence of remote writes. Their study shows that remote writes improve the performance of these synchronization algorithms significantly.

Finally Michael and Scott [18] have studied the performance impact of different implementations of atomic instructions in scalable multiprocessors. However, their study focuses on the atomic primitives rather than on the algorithms that use them.

6 Conclusions

In this paper we have studied the running time and communication behavior of several lock, barrier, reduction, broadcasting, and task queue implementations on top of invalidate and update-based protocols on a scalable multiprocessor. Our analysis indicates that locks can profit from update-based protocols, especially for small to medium contention levels. In addition, our results show that scalable barriers and logarithmic broadcasting strategies can benefit greatly from these protocols, independently of the number of processors. Our reduction experiments demonstrate that sequential reductions also benefit from protocols based on updates, but their performance is only competitive under heavy contention. Finally, our results show that update-based protocols achieve poor performance for task queues, even when distributed queues are utilized.

Our experience and findings have several implications for parallel programming and scalable multiprocessor design:

- Implementations that exhibit a low degree of sharing and short write runs achieve excellent performance under update-based protocols regardless of the number of processors. Multiprocessors with a hard-wired update-based protocol must then be programmed carefully, so that the implementation of each programming construct approximates this sharing behavior as much as possible. Our update-conscious MCS lock, for instance, was designed to reduce the degree of sharing of the MCS lock.

- For constructs that do not admit implementations with such a well-behaved sharing pattern, the update-based protocols are not always the best choice, especially for large numbers of processors. Thus, none of the coherence protocols we studied is ideal for all constructs/implementations. This means that multiprocessors that support a single coherence protocol are bound to be inefficient in certain cases. In contrast, multiprocessors with programmable protocol processors should achieve good performance consistently, provided that the best combination of implementation and protocol is chosen for each construct.

- Most of our construct implementations could be improved by selectively using producer-initiated communication. This type of communication is well-suited to producer-consumer data or migratory data that migrate to predictable destinations. However, producer-initiated communication requires the support of the underlying system (which we did not assume in this work), either through hardware primitives such as remote write [1] or through fast user-level access to the messaging hardware as in Alewife [2]. Remote writes can improve performance by allowing the writing processor to update another processor's cache or the memory, possibly dropping the copy of the written block from the writer's cache. When applicable, this achieves the goal of update-based protocols, namely to avoid sharing misses, without generating any useless traffic. Regular messages can be used for the same types of data as remote writes, but usually require coarser-grain sharing to amortize their cost. Thus, the availability of either fine or coarse-grain producer-initiated communication mechanisms in current and future multiprocessors is certainly justified, at least in terms of the implementation of parallel programming constructs.

In summary, this study shows that the implementation of parallel programming idioms for scalable multiprocessors must take the coherence protocol into account, since invalidate and update-based protocols sometimes lead to different design decisions. Programmers of update-based multiprocessors and machines with protocol processors should then carefully implement their constructs if applications are to avoid unnecessary overheads.

References

1. H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An Evaluation of Fine-Grained Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

2. A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, June 1995.

3. J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

4. R. Bianchini and L. I. Kontothanassis. Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update-Based Coherence Protocols. In *Proceedings of the 28th Annual Simulation Symposium*, pages 115–124, April 1995.

5. R. Bianchini, T. J. LeBlanc, and J. E. Veenstra. Categorizing Network Traffic in Update-Based Protocols on Scalable Multiprocessors. In *Proceedings of the International Parallel Processing Symposium '96*, pages 142–151, April 1996.

6. F. Dahlgren, M. Dubois, and P. Stenström. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 187–197, April 1994.

7. M. Dubois, J. Skeppstedt, and P. Stenström. Essential Misses and Data Traffic in Coherence Protocols. *Journal of Parallel and Distributed Computing*, 29(2):108–125, September 1995.

8. M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 88–97, May 1993.

9. S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 377–381, August 1991.

10. S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–383, May 1988.

11. A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transaction on Computers*, 41(7):794–810, July 1992.

12. C. Holt, J. P. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 134–145, May 1996.

13. Kendall Square Research Corp. *KSR1 Principles of Operation.* Kendall Square Research Corporation, 1992.

14. J. Kuskin *et al.* The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

15. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

16. D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

17. J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

18. M. M. Michael and M. L. Scott. Implementation of General-Purpose Atomic Primitives for Distributed Shared-Memory Multiprocessors. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 222–231, January 1995.

19. T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, October 1992.

20. S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

21. J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

22. J. E. Veenstra and R. J. Fowler. A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols. In *Proceedings of the 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 149–157, October 1992.

23. J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201–207, January 1994.

24. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, May 1995.

List of Figures

```
type lock = record
    next_ticket : unsigned integer := 0
    now_serving : unsigned integer := 0

procedure acquire_lock (L : ^lock)
    my_ticket : unsigned integer := fetch_and_increment (&L->next_ticket)
    repeat while L->now_serving != my_ticket
        pause(my_ticket - L->now_serving)

procedure release_lock (L : ^lock)
    L->now_serving := L->now_serving + 1
```

Figure 1: The ticket lock with proportional backoff [17].

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
    I->next := nil
    predecessor : ^qnode := fetch_and_store (L, I)
    if predecessor != nil
        // queue was non-empty
        I->locked := true
        predecessor->next := I
        // Flush *pred in update-conscious MCS
        repeat while I->locked

procedure release_lock (L : ^lock, I: ^qnode)
    if I->next = nil
        // no known successor
        if compare_and_swap (L, I, nil)
            return
        repeat while I->next = nil
    I->next->locked := false
    // Flush *(I->next) in update-conscious MCS
```

Figure 2: The MCS lock [17].

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true

procedure central_barrier
    // each processor toggles its own sense
    local_sense := not local_sense
    if fetch_and_decrement (&count) = 1
        count := P
        // last processor toggles global sense
        sense := local_sense
    else
        repeat until sense = local_sense
```

Figure 3: The sense-reversing centralized barrier [17].

```
type flags = record
    myflags : array [0..1][0..LogP-1] of Boolean
    partflags : array [0..1][0..LogP-1] of ^Boolean

processor private parity : integer := 0
processor private sense : Boolean := true
processor private localflags : ^flags
shared allnodes : array [0..P-1] of flags

// on proc i, localflags points to allnodes[i]
// initially allnodes[i].myflags[r][k] is false for all i, r, k
// if j = (i+2^k) % P, then for r = 0, 1:
//     allnodes[i].partflags[r][k] points to allnodes[j].myflags[r][k]

procedure dissemination_barrier
    for i : integer := 0 to LogP-1
        localflags^.partflags[parity][i]^ := sense
        repeat until localflags^.myflags[parity][i] = sense
    if parity = 1
        sense := not sense
    parity := 1 - parity
```

Figure 4: The dissemination barrier [17].

```
type treenode = record
    parentpointer : ^Boolean
    havechild, childnotready : array [0..3] of Boolean
    dummy : Boolean     // pseudo-data

    shared nodes : array [0..P-1] of treenode
    processor private vpid : integer
    processor private sense : Boolean
    shared globalsense: Boolean

    // on proc i, sense is initially true
    // in nodes[i]: havechild[j] = true if 4*i+j < P; otherwise false
    //   parentpointer = &nodes[floor((i-1)/4)].childnotready[(i-1)%4],or &dummy if i=0
    //   initially childnotready = havechild, sense = true, globalsense = false

    procedure tree_barrier
        with nodes[vpid] do
            repeat until childnotready = {false, false, false, false}
            childnotready := havechild    // prepare for next barrier
            parentpointer^ := false       // let parent know I'm ready
            if vpid != 0  // if not root, wait until the root signals wakeup
                repeat until globalsense = sense
            else
                globalsense := sense
            sense := not sense
```

Figure 5: The tree-based barrier [17].

```
shared sum : integer := 0
shared local_sum : array [0..P-1] of integer
shared Barrier : barrier
processor private pid, i, tmp : integer

// Code that changes local_sum[pid]
BARRIER(Barrier)
    if pid = 0
        tmp := sum
        for i := 0 until i = P-1
            tmp := tmp + local_sum[i]
            if tmp > LIMIT
                tmp := 0
        sum := tmp
BARRIER(Barrier)
// Code that uses sum
```

Figure 6: Sequential reduction operation.

```
shared sum : integer := 0
shared Lock : lock
shared Barrier : barrier
processor private local_sum : integer

// Code that changes local_sum
LOCK(Lock)
    sum := sum + local_sum
    if sum > LIMIT
        sum := 0
UNLOCK(Lock)
BARRIER(Barrier)
    // Code that uses sum
BARRIER(Barrier)
```

Figure 7: Parallel reduction operation.

```
type buffer = record
    data : array [0..DATA_SIZE] of integer
    index, counter : integer

shared broad : array[0..P-1][0..NUM_BUFFS-1] of buffer
processor private pid, ind, buffer : integer

// Proc 0 is the fixed producer of the data.
// Ind stores the produced item's order number.

procedure bcast_linear
    if pid != 0
        buffer := ind % NUM_BUFFS
        repeat until broad[0][buffer].index = ind
        // Consume the data fetching it from the producer
        fetch_and_decrement (&broad[0][buffer].counter)
        ind := ind + 1
    else
        ind := ind + 1
        buffer := ind % NUM_BUFFS
        repeat until broad[0][buffer].counter = 0
        // Produce new data item and store it in broad[0][buffer].data
        broad[0][buffer].counter := P-1
        // Fence
        broad[0][buffer].index := ind
```

Figure 8: Linear broadcasting.

```
type buffer = record
    data : array [0..DATA_SIZE] of integer
    index, counter : integer

shared broad : array[0..P-1][0..NUM_BUFFS-1] of buffer
processor private pid, parent, children, ind, buffer : integer

// Broadcast tree is set up during initialization. Proc 0 is at the root.
// Parent stores the id of the parent in the broadcasting tree.
// Children stores the number of children each processor has.
// Ind stores the produced item's order number.

procedure move_data
    repeat until broad[parent][buffer].index = ind
    repeat until broad[pid][buffer].counter = 0
    // Copy data from broad[parent][buffer].data to broad[pid][buffer].data
    broad[pid][buffer].counter := children
    // Fence
    broad[pid][buffer].index := ind
    fetch_and_decrement (&broad[parent][buffer].counter)

procedure bcast_consumer
    if pid != 0
        buffer := ind % NUM_BUFFS
        move_data
        // Consume the new data
        ind := ind + 1
    else
        ind := ind + 1
        buffer := ind % NUM_BUFFS
        repeat until broad[0][buffer].counter = 0
        // Produce new data item and store it in broad[0][buffer].data
        broad[0][buffer].counter := children
        // Fence
        broad[0][buffer].index := ind
```

Figure 9: Consumer-driven logarithmic broadcasting.

```
type buffer = record
    data : array [0..DATA_SIZE] of integer
    produced, consumed : integer

shared broad : array[0..P-1][0..NUM_BUFFS-1] of buffer
processor private pid, ind, buffer, i : integer
processor private children : array[0..P-1] of integer

// Broadcast tree is set up during initialization. Proc 0 is at the root.
// Ind stores the produced item's order number.
// Children stores the pid of each child processor.

procedure move_data (child : integer)
    repeat until broad[pid][buffer].produced = 1
    repeat until broad[child][buffer].consumed = 1
    // Copy data from broad[pid][buffer].data to broad[child][buffer].data
    broad[child][buffer].consumed := 0
    // Fence
    broad[child][buffer].produced := 1

procedure bcast_producer
    i := 0
    repeat until (i = DEGREE or children[i] = -1)
        if children[i] != -1
            move_data (children[i])
        i := i + 1
    if pid != 0
        buffer := ind % NUM_BUFFS
        repeat until broad[pid][buffer].produced = 1
        // Consume the new data
        broad[pid][buffer].produced := 0
        broad[pid][buffer].consumed := 1
        ind := ind + 1
    else
        ind := ind + 1
        buffer := ind % NUM_BUFFS
        // Produce new data item and store it in broad[0][buffer].data
        // Fence
        broad[0][buffer].produced := 1
```

Figure 10: Producer-driven logarithmic broadcasting.

```
shared queue : integer := 0
shared Barrier : barrier

procedure cent_task_queue
    my_task : integer
    round : integer := 0
    group_tasks : integer := TOT_TASKS / P / 2
    repeat until round = TOT_ROUNDS
        my_task := fetch_and_increment (&queue)
        my_task := my_task * group_tasks
        if my_task > round * TOT_TASKS
            round := round + 1
            BARRIER(Barrier)
        my_task := my_task % TOT_TASKS
        // Code that uses my_task work descriptor
```

Figure 11: Centralized task queue-based computation.

```
shared queue : array [0..P-1] of integer
shared Lock : array [0..P-1] of lock
shared Barrier : barrier

procedure dist_task_queue
    my_task, i, remote : integer
    round : integer
    local_tasks : integer := TOT_TASKS / P
    for round := 1 to TOT_ROUNDS
        // Perform local tasks
        do
            LOCK(Lock[pid])
                if queue[pid] < round * local_tasks
                    my_task := queue[pid]
                    queue[pid] := queue[pid] + 1
                else
                    my_task := -1
            UNLOCK(Lock[pid])
            if my_task != -1
                // Code that uses my_task work descriptor
        until my_task == -1
        // Search for remote tasks to execute in 1/4 of the processors
        for i := 1 to P-1 step 4
            remote := (pid + i) % P  // Find next proc in round-robin fashion
            repeat while queue[remote] < round * local_tasks - BEHIND // Hint
                LOCK(Lock[remote])
                    if queue[remote] < round * local_tasks - BEHIND + 1
                    // Remote proc is indeed late
                        my_task := queue[remote]
                        queue[remote] := queue[remote] + 1
                    else
                        my_task := -1
                UNLOCK(Lock[remote])
                if my_task != -1
                    // Code that uses my_task work descriptor
        BARRIER(Barrier)
```

Figure 12: Distributed task queue-based computation.

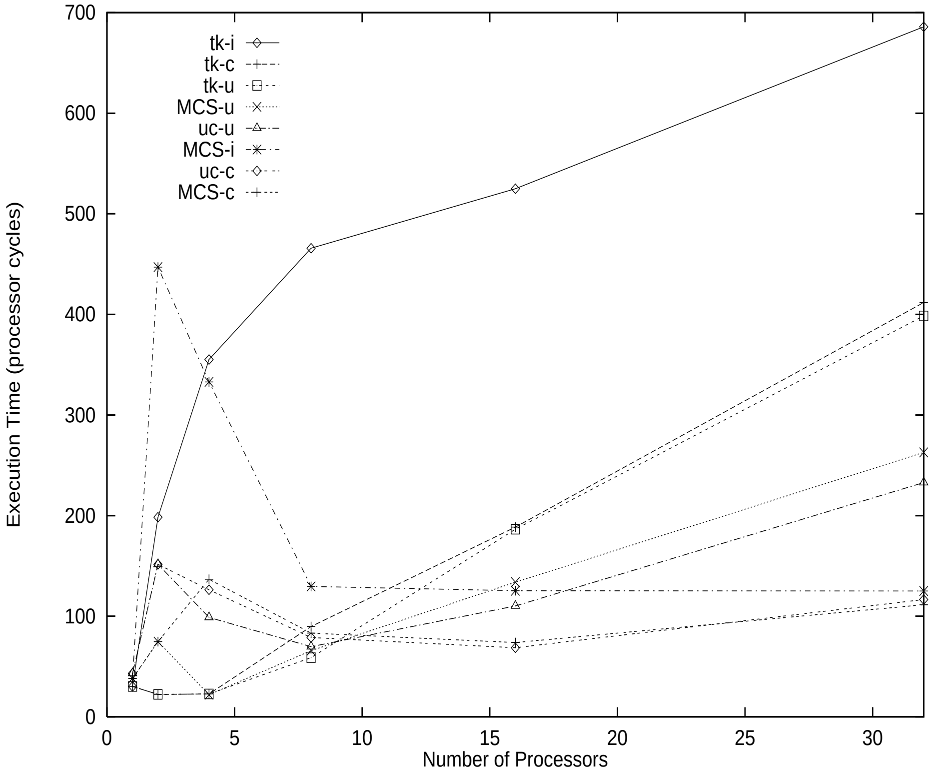

Figure 13: Performance of spin locks in synthetic program.

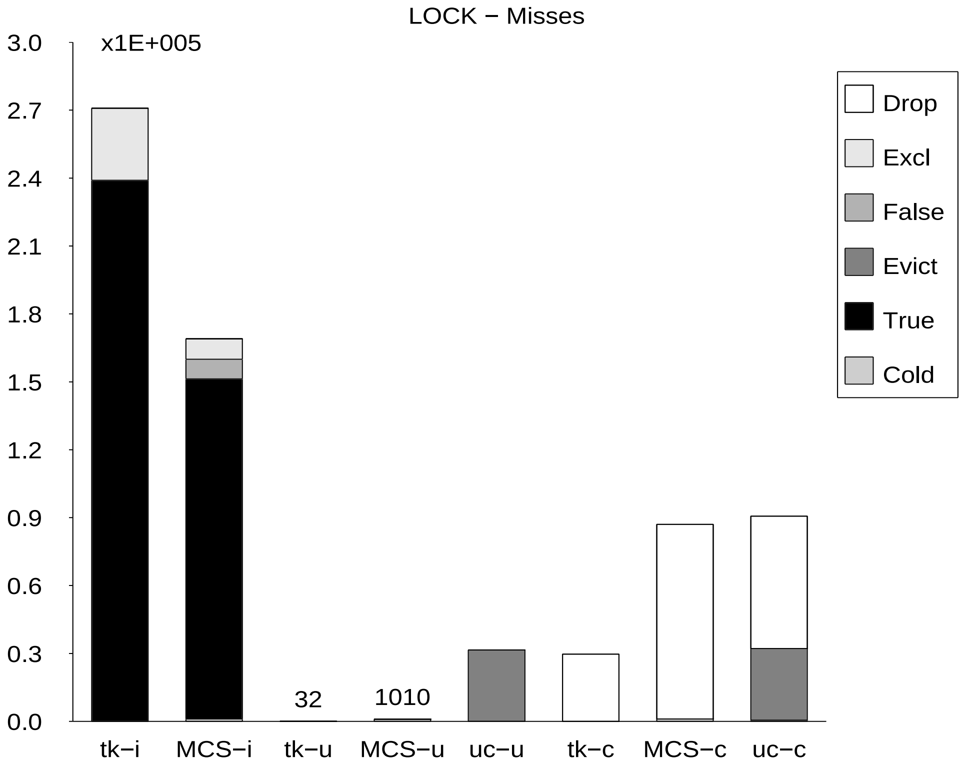

Figure 14: Miss traffic of spin locks in synthetic program.

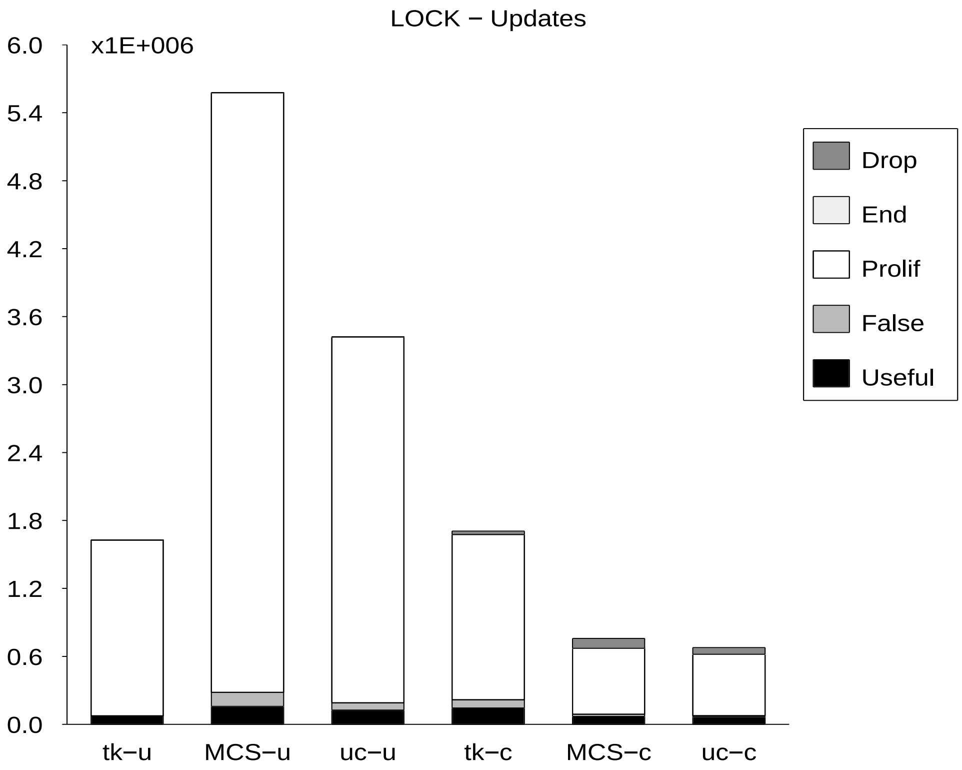

Figure 15: Update traffic of spin locks in synthetic program.

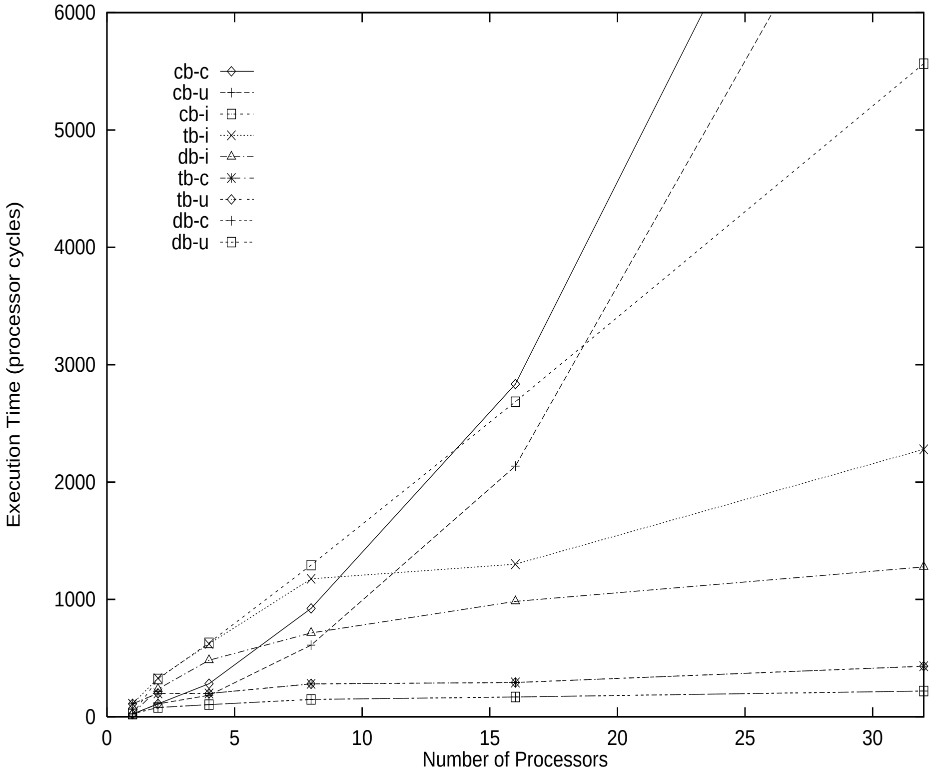

Figure 16: Performance of barriers in synthetic program.

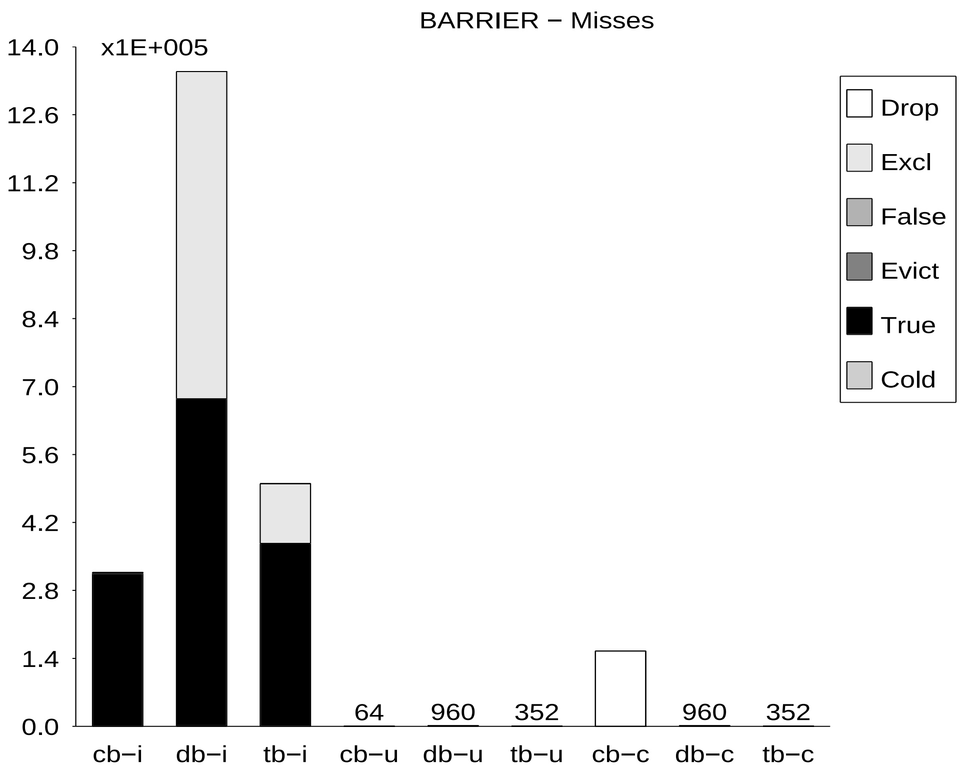

Figure 17: Miss traffic of barriers in synthetic program.

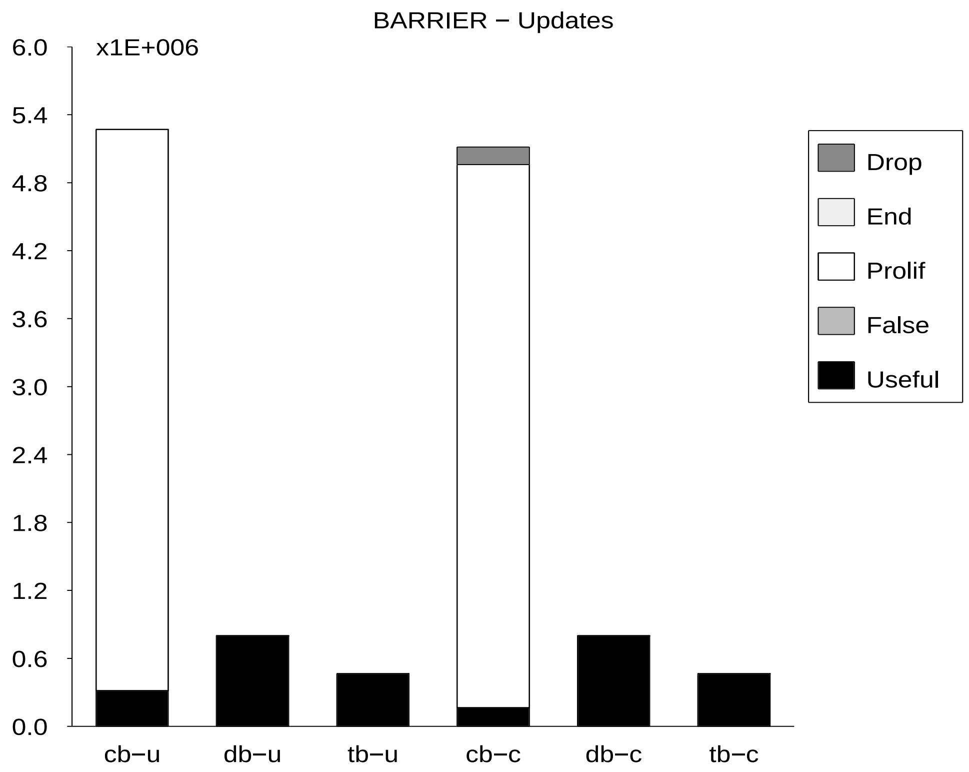

Figure 18: Update traffic of barriers in synthetic program.

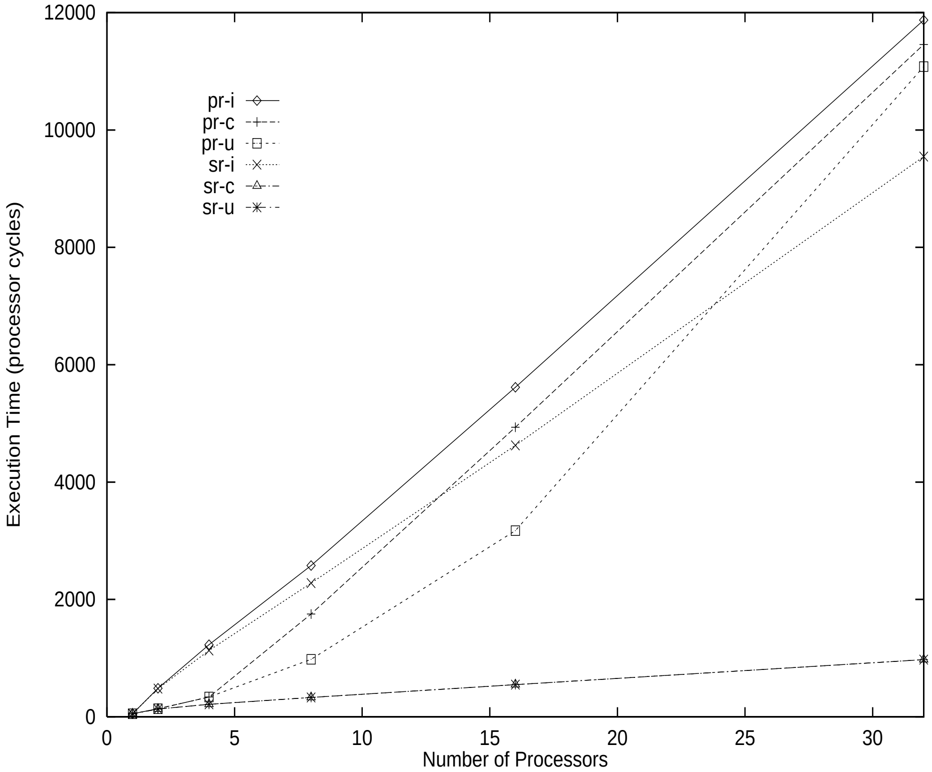

Figure 19: Performance of reductions in synthetic program.

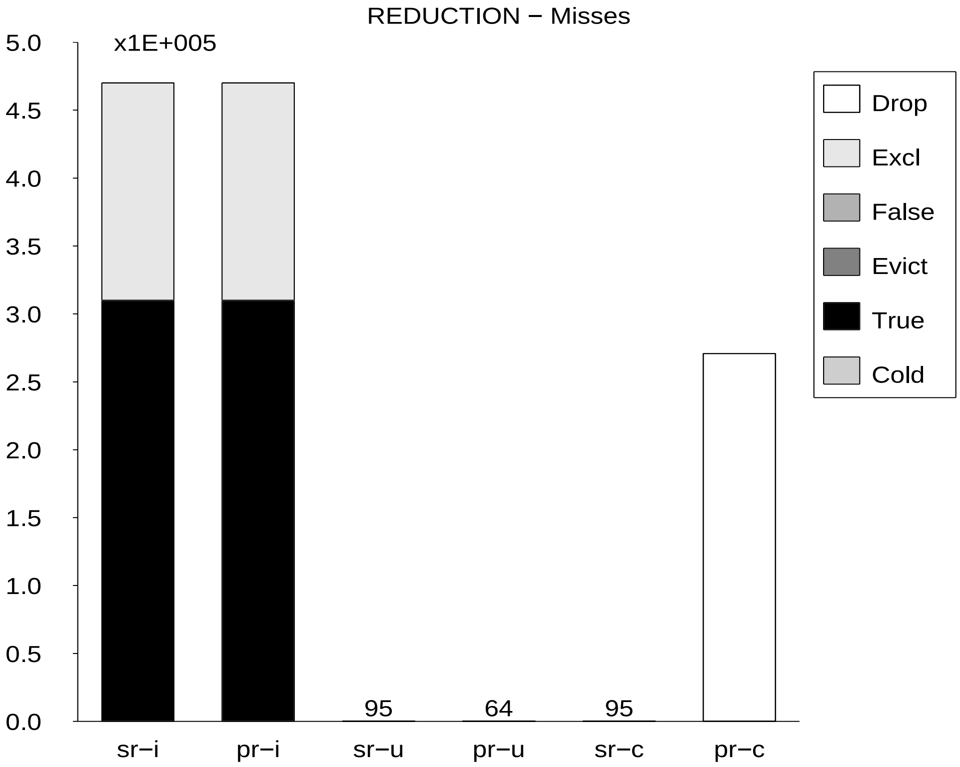

Figure 20: Miss traffic of reductions in synthetic program.

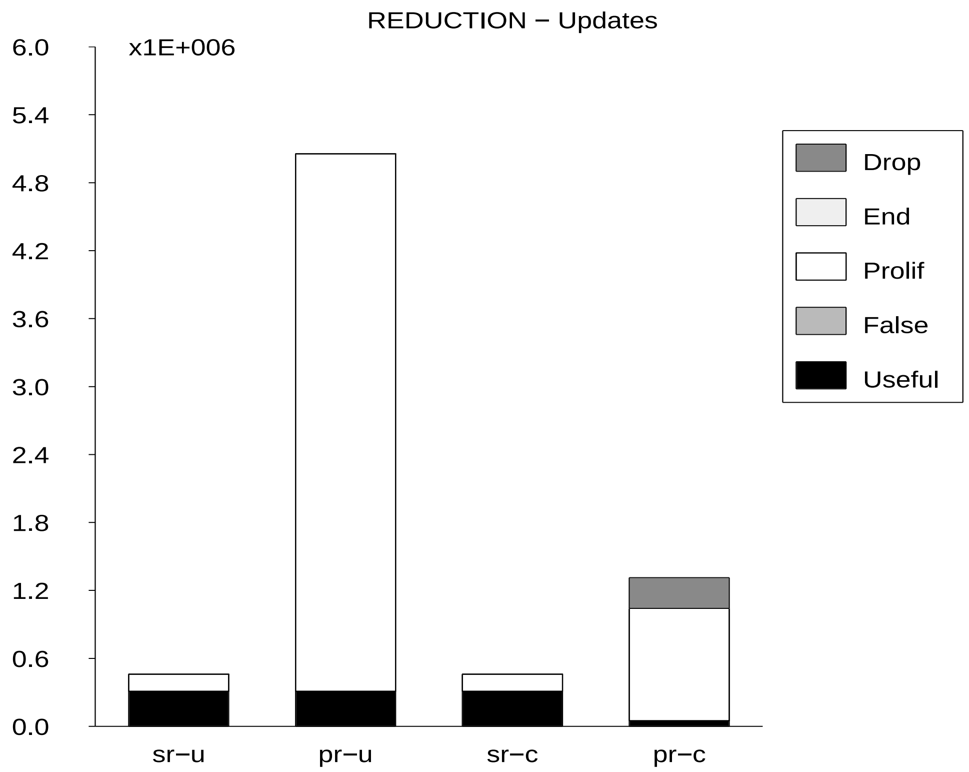

Figure 21: Update traffic of reductions in synthetic program.

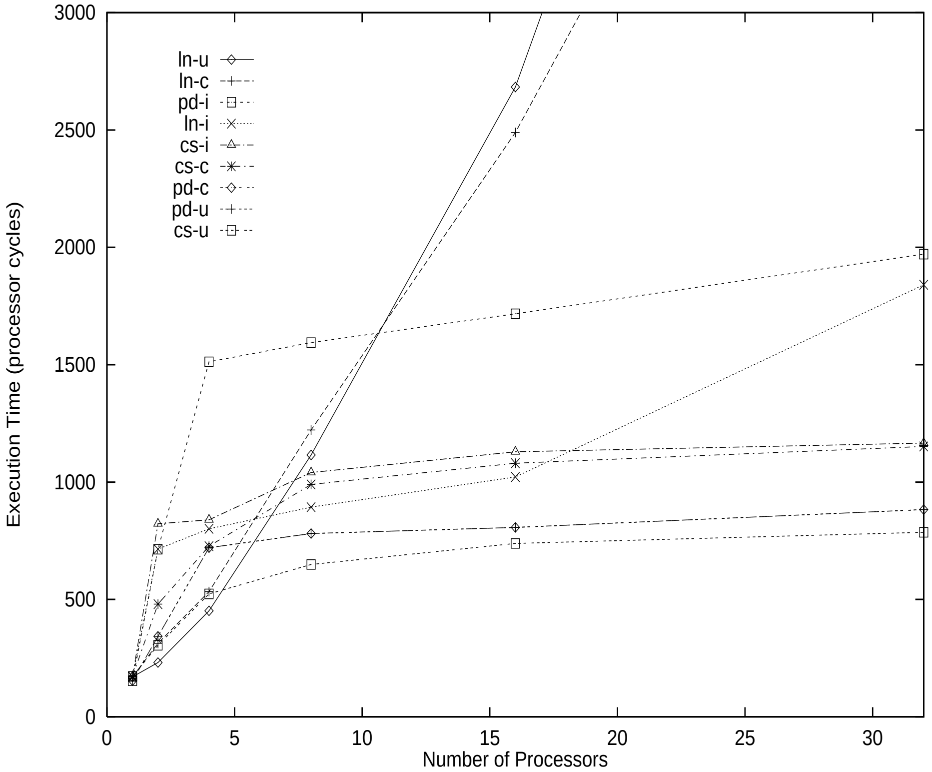

Figure 22: Performance of broadcasting in synthetic program.

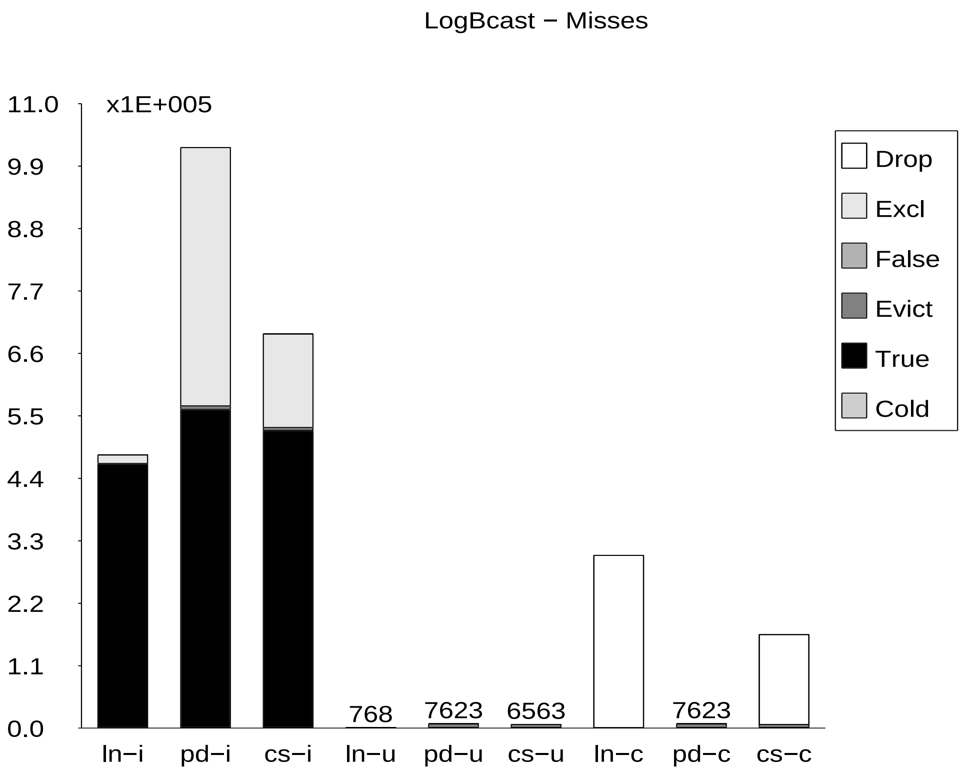

Figure 23: Miss traffic of broadcasting in synthetic program.

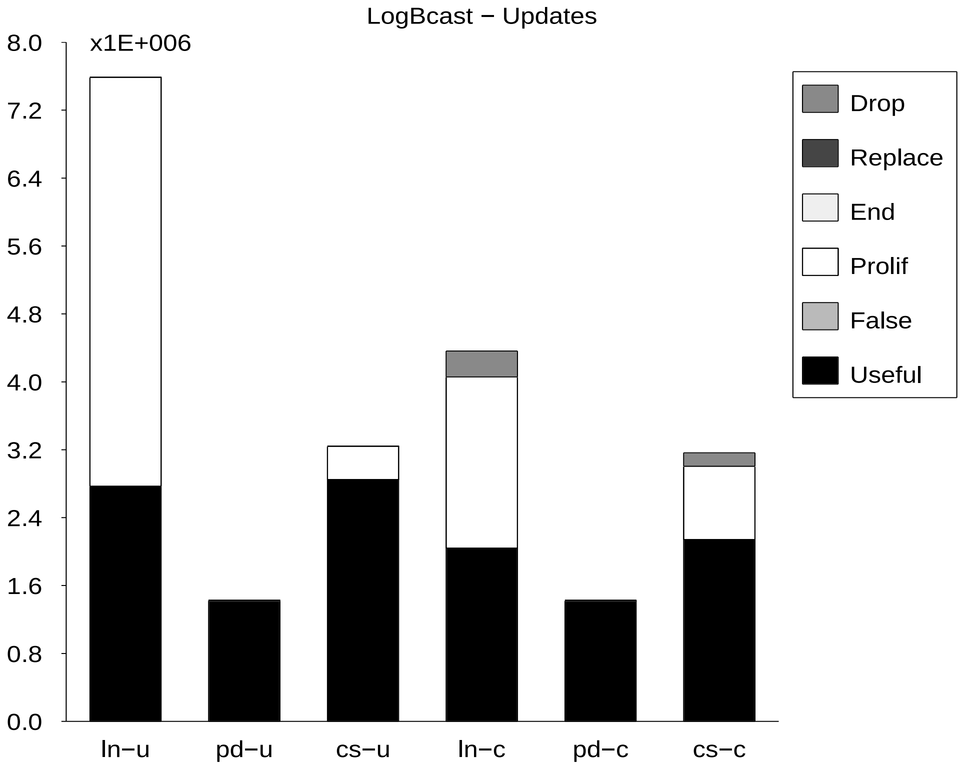

Figure 24: Update traffic of broadcasting in synthetic program.

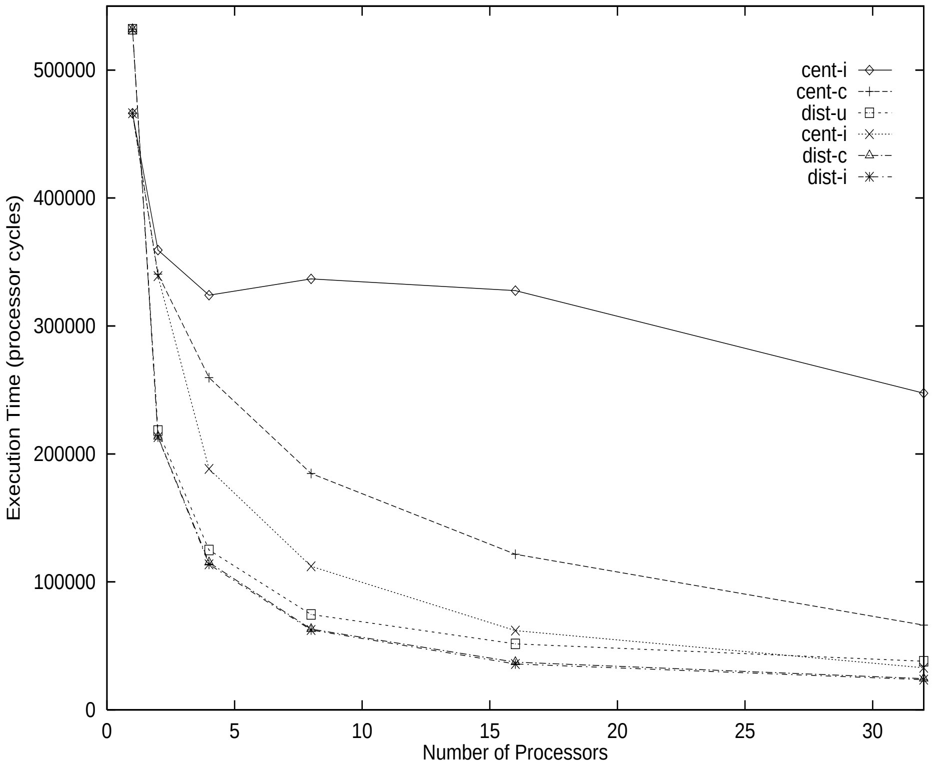

Figure 25: Performance of task queue in synthetic program.

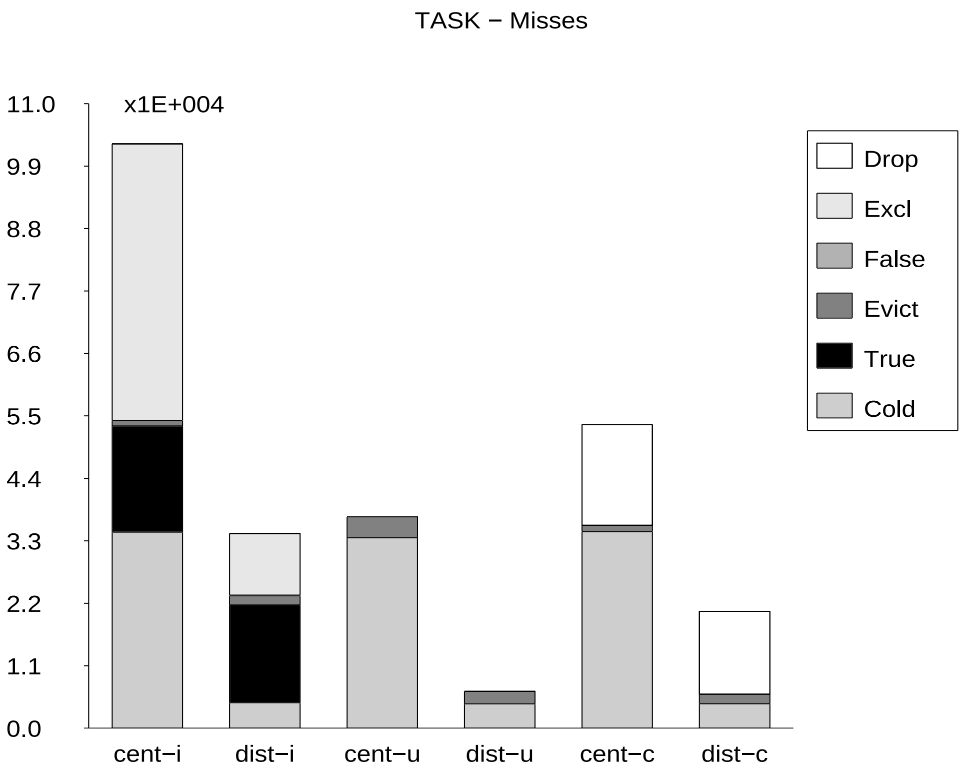

Figure 26: Miss traffic of task queue in synthetic program.

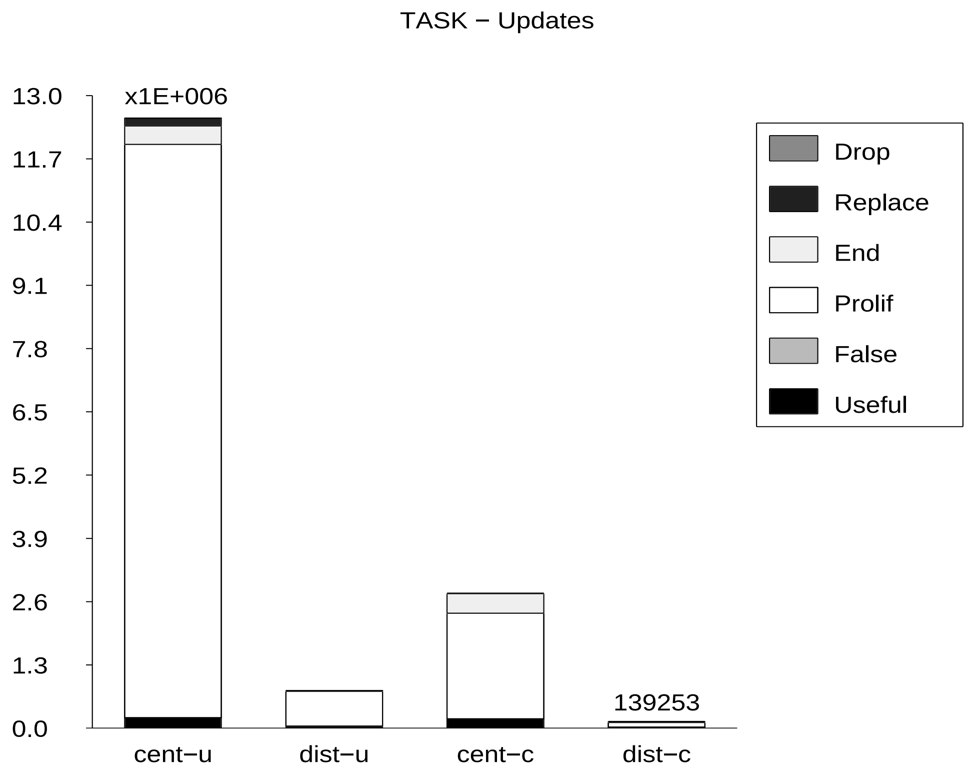

Figure 27: Update traffic of task queue in synthetic program.